Reference Manual of Impulse System Calls [1]

Lixin Zhang, Leigh Stoller
E-mail: {lizhang,stoller}@cs.utah.edu

UUCS-99-018

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

July 9, 1999

Abstract

This document describes the Impulse system calls. The Impulse system calls allow user applications to use remapping functionality provided by the Impulse Adaptive Memory System to remap their data structures. Impulse supports several remapping algorithms. User applications choose the desired remapping algorithms by calling the right Impulse system calls. This note uses detailed examples to illustrate each Impulse system call.

[1]This effort was sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number F30602-98-1-0101 and DARPA Order Numbers F393/00-01 and F376/00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official polices or endorsements, either express or implied, of DARPA, AFRL, or the US Government.

Contents

1 Introduction

This document describes the Impulse system calls. The Impulse system calls and their related data structures are defined in the following header file.

`/home/css/impsrc/lizhang/simpulse/kernel/amssup.h`

User applications should include this header file and link with the following object file to pick up the Impulse system calls.

`/home/css/impsrc/lizhang/simpulse/kernel/libkernel.o`

Currently, the Impulse system calls implement the following optimizations: **no-copy superpage formation**, **scatter/gather strided data structure**, **no-copy matrix transpose**, **no-copy page coloring**, and **scatter/gather through an indirection vector**. Depending on what kind of values are stored in the indirection vector, **scatter/gather through an indirection vector** is further split into two forms: **scatter/gather through an index vector**, if the indirection vector stores indices to a data vector; **scatter/gather through an offset vector**, if the indirection vector stores byte offsets.

2 System Call Interface

2.1 ams_mapshadow()

```
int
ams_mapshadow(int       type,
              void     *datablock);
```

`ams_mapshadow` initializes the data structures and configures the Impulse main memory controller necessary for performing the optimization specified by `type`. `datablock` is the address of a type-specific argument block. The six types currently supported are `AMS_TYPE_SUPERPAGE`, `AMS_TYPE_BASESTRIDE`, `AMS_TYPE_TRANSPOSE`, `AMS_TYPE_VINDIRECT`, `AMS_TYPE_PAGECOLOR`, and `AMS_TYPE_VINDIRECT2`. They are described in the following subsections.

2.1.1 ams_mapshadow(AMS_TYPE_SUPERPAGE, datablock)

```
typedef struct {
        unsigned  vaddr;
        int       size;
        int       prefcount;
        int       prefinfo;
} superpage_args_t;
```

This call implements **no-copy superpage formation** remapping. It maps the virtual memory region, with `size` bytes in length, at address `vaddr` into an equally sized contiguous shadow memory region. The MTLB will translates the new shadow memory region to the physical pages to which the original virtual memory region maps. Both `vaddr` and `size` must be page-aligned values. `prefcount` is the number of lines to prefetch ahead; and `prefinfo` indicates prefetch direction and

prefetch stride. For example, if the Impulse memory controller serviced a load request for a cache line at address **A** when `prefcount` equals 2 and `prefinfo` equals -3, it would prefetch *two* cache lines whose addresses are (**A** − *3* * cache line size) and (**A** − *6* * cache line size).

Since the region is contiguous in both virtual memory and shadow memory, it can be mapped using superpage TLB entries. This call converts the virtual memory region to TLB superpages, with the largest possible superpages allocated, based on the alignment of `vaddr`. Because superpages must be aligned to their sizes, superpages are allocated by walking the virtual address region and assigning the largest possible superpage restricted by the current alignment. For example, if the current address is 16Kbyte aligned, a 16Kbyte superpage can be assigned. The address is then incremented; and the new address alignment is checked. This procedure proceeds until the end of the region is reached.

Superpage sizes are configured in the kernel, and are powers of 2 multiple of base pages ranging from 8K to 4M bytes in size. The normal page size is 4K bytes. In practice, the first few and last few pages are often normal pages, with the pages in the middle being larger. For example, a virtual memory region at 0x00039000 with 0x100000 bytes will be converted to 9 normal pages or superpages with 4K, 8K, 16K, 256K, 512K, 128K, 64K, 32K, and 4K bytes respectively. This conversion will reduce the number of TLB entries required for this region from 256 to 9.

`ams_mapshadow` returns 0 on success and -1 otherwise. Figure 1 is a simple example using `ams_mapshadow(AMS ...)`.

ERRORS:

`EINVAL` Either `vaddr` or `size` is not page-aligned.

2.1.2 ams_mapshadow(AMS_TYPE_BASESTRIDE, datablock)

```
typedef struct {
        unsigned *newaddr;
        unsigned  vaddr;
        int       count;
        int       objsize;
        int       stride;
        int       offset;
        int       prefcount;
        int       prefinfo;
        int       salign;
        int       valign;
} basestride_args_t;
```

The type `AMS_TYPE_BASESTRIDE` initializes the data structures necessary for implementing **scatter/gather strided data structure**. The argument `datablock` should point to the address of a `basestride_args_t` structure.

`newaddr` should be a pointer to a location in the application where the kernel will store the virtual address of the newly allocated region which contains only required objects. `vaddr` is the page-aligned virtual address of the original vector. The original vector contains `count` number of strides, each of which is `stride` bytes in length. Obviously, the size of the original vector is `count` *

```
/*
 * Creates superpages for matrix A
 */

#define PAGESIZE 0x1000

double foo(double *A, int size)
{
   int     i;
   double  sum = 0;
   superpage_args_t sp_args;

#ifdef IMPULSE
   sp_args.vaddr     = (unsigned) A;
   sp_args.size      = size;
   sp_args.prefcount = 1;
   sp_args.prefinfo  = 1;

   if (ams_mapshadow(AMS_TYPE_SUPERPAGE, &sp_args) < 0) {
      printf("ams_mapshadow_superpage failed\n");
      exit(1);
   }
#endif

   for (i = 0; i < size; i += PAGESIZE)
      sum += A[i];

   return sum;
}
```

Figure 1: Code fragment illustrating `ams_mapshadow(AMS_TYPE_SUPERPAGE, ...)` usage.

stride bytes. The required object is objsize bytes in length; and its byte offset from the base of the stride is give by offset. prefcount is the number of lines to prefetch ahead; and prefinfo indicates prefetch direction and prefetch stride.

This function creates a new vector, whose size is count * objsize bytes. From the CPU's perspective, the new vector contains just the required objects. This new vector is mapped to an equally sized region of shadow address space, but no real pages of physical memory are allocated. salign is the expected alignment of the new vector's shadow address space; and valign is the expected alignment of the new vector's virtual address space.

Once the new vector has been created, the necessary information is downloaded to the memory controller.

ams_mapshadow returns 0 on success, and -1 otherwise. If successful, the virtual address of the new vector is placed into the memory location pointed to by newaddr. Figure 2 is a simple code fragment illustrating the use of ams_mapshadow(AMS_TYPE_BASESTRIDE, ...).

ERRORS:

EINVAL	vaddr is not paged aligned.
EFAULT	newaddr specifies an invalid address.

2.1.3 ams_mapshadow(AMS_TYPE_TRANSPOSE, datablock)

```
typedef struct {
        unsigned *newaddr;
        unsigned  vaddr;
        int       elemsize;
        int       rownum;
        int       rowsize;
        int       prefcount;
        int       prefinfo;
        int       salign;
        int       valign;
} transpose_args_t;
```

The type AMS_TYPE_TRANSPOSE is used by remapping algorithm **no-copy matrix transpose**. This system call maps a two-dimension matrix to its transpose without copying. The argument datablock should point to the address of a transpose_args_t structure.

newaddr should be a pointer to a location in the application where the kernel will store its return value. vaddr is the page-aligned virtual address of the original matrix, whose element is elemsize bytes in length. The original matrix contains rownum number of rows, each of which has rowsize bytes. prefcount indicates how many lines to prefetch ahead; and prefinfo indicates the prefetch direction and prefetch stride.

The return value of this function is the virtual address of a new matrix – the transpose of the original matrix. This new matrix is mapped to an equally sized region of shadow address space, but no real pages of physical memory are allocated. salign and valign specifies the required alignment of the new matrix's shadow address region and virtual address region.

```
/*
 * Compute the sum of A[8*i+2], where i is from 0 to (size/8 - 1).
 */
float foo(float *A, int size)
{
   basestride_args_t  bs_args;
   float              *Anew;
   float              sum  = 0;
   int                step = 8;

#ifdef IMPULSE
   bs_args.newaddr   = &Anew;
   bs_args.vaddr     = A;
   bs_args.count     = size / step;
   bs_args.objsize   = sizeof(float);
   bs_args.stride    = sizeof(float) * step;
   bs_args.offset    = sizeof(float) * 2;
   bs_args.prefcount = 1;
   bs_args.prefinfo  = 1;
   bs_args.salign    = 0x4000;
   bs_args.valign    = 0x2000;

   if (ams_mapshadow(AMS_TYPE_BASESTRIDE, &bs_args) == -1) {
      perror("ams_mapshadow failed.");
      exit(1);
   }
   for (i = 0; i < size / step; i++)
      sum += Anew[i];
#else /* Non-Impulse version */
   for (i = 0; i < size / step; i++)
      sum += A[i * 8 + 2];
#endif
   return sum;
}
```

Figure 2: Code fragment illustrating `ams_mapshadow(AMS_TYPE_BASESTRIDE, ...)` usage.

Once the new matrix has been created, the necessary information is downloaded to the memory controller.

ams_mapshadow returns 0 on success, and -1 otherwise. If successful, the virtual address of the new matrix is placed into the memory location pointed to by newaddr. Figure 3 is a simple code fragment illustrating the use of ams_mapshadow(AMS_TYPE_TRANSPOSE, ...).

ERRORS:

EINVAL	vaddr is not paged aligned.
EFAULT	newaddr specifies an invalid address.

2.1.4 ams_mapshadow(AMS_TYPE_VINDIRECT, datablock)

```
typedef struct {
        unsigned *newaddr;
        unsigned  vaddr;
        int       count;
        int       objsize;
        unsigned  iv_vaddr;
        int       iv_count;
        int       iv_objsize;
        int       isfortran;
        int       maxcount;
        int       prefcount;
        int       prefinfo;
        int       salign;
        int       valign;
} vindirect_args_t;
```

The call to ams_mapshadow(AMS_TYPE_VINDIRECT, ...) initializes the data structures necessary for **scatter/gather through an indirection vector**, where the indirection vector stores indices into the original data vector. The argument datablock should point to the address of a vindirect_args_t structure. newaddr should be a pointer to a location in the application where the kernel will store its return value.

vaddr is the page-aligned virtual address of the original data vector, which contains count number of objects and whose object is objsize bytes in length. iv_vaddr is the page-aligned virtual address of the original indirection vector, which contains iv_count number of objects and whose object is iv_objsize bytes in length. isfortran indicates whether or not the indirection vector stores Fortran-style array subscripts, i.e., subscripts starting from 1, not 0. The indirection vector is copied into a newly allocated region of *contiguous physical* memory; and then its virtual region is remapped to the new physical memory. The original physical pages are returned to the system. prefcount is the number of lines to prefetch ahead; and prefinfo indicates the prefetch direction and prefetch stride.

The return value of this function is the virtual address of a new vector, whose size is maxcount * objsize bytes. This new vector is mapped to an equally sized region of shadow address space, but no real pages of physical memory are allocated. The alignment of this new vector's shadow

```
/*
 * Dense matrix-matrix multiplication: C = A * B.
 * A, B, and C are (size x size) matrices.
 */
foo(double *A, double *B, double *C, int size)
{
   transpose_args_t  tr_args;
   double            *Bnew, sum;
   int               i, j, k;

#ifdef IMPULSE
   tr_args.newaddr   = (unsigned *) &Bnew;
   tr_args.vaddr     = (unsigned) x;
   tr_args.elemsize  = sizeof(double);
   tr_args.rownum    = size;
   tr_args.rowsize   = sizeof(double) * size;
   tr_args.prefcount = 1;
   tr_args.prefinfo  = 1;
   tr_args.salign    = 0; /* dont' care */
   tr_args.valign    = 0; /* dont' care */

   if (ams_mapshadow(AMS_TYPE_TRANSPOSE, &tr_args) == -1) {
      perror("ams_mapshadow failed.");
      exit(1);
   }
#endif

   for (i = 0; i < size; i++)
      for (j = 0; j < size; j++) {
         for (sum = 0, k = 0; k < size; k++)
#ifdef IMPULSE
            sum += A[i][k] * Bnew[j][k];
#else
            sum += A[i][k] * B[k][j];
#endif
         C[i][j] = sum;
      }
}
```

Figure 3: Code fragment illustrating `ams_mapshadow(AMS_TYPE_TRANSPOSE, ...)` usage.

memory region is specified by salign; and the alignment of its virtual memory region is specified by valign.

Once the new vector has been created, the necessary information is downloaded to the memory controller.

ams_mapshadow returns 0 on success, and -1 otherwise. If successful, the virtual address of the new vector is placed into the memory location pointed to by newaddr. Figure 4 is a simple code fragment illustrating the use of ams_mapshadow(AMS_TYPE_VINDIRECT, ...).

ERRORS:

EINVAL	Either vaddr or iv_vaddr is not paged aligned.
EFAULT	newaddr specifies an invalid address.

2.1.5 ams_mapshadow(AMS_TYPE_PAGECOLOR, datablock)

```
typedef struct {
        unsigned  vaddr;
        int       size;
        int       waysize;
        int       colorfactor;
        int       colorid;
        int       prefcount;
        int       prefinfo;
} pagecolor_args_t;
```

The type AMS_TYPE_PAGECOLOR maps the specified virtual region to the designated color in physically-indexed L2 cache. The argument datablock should point to the address of a pagecolor_args_t structure.

vaddr is the page-aligned starting virtual address of the remapped region which has size bytes. waysize represents the way size of L2 cache, which is the L2 cache size divided by its associativity. colorfactor is number of colors we want the L2 cache to have. colorid indicates the color to which the region will solely map. prefcount indicates how many lines to prefetch ahead; and prefinfo indicates the prefetch direction and prefetch stride.

Once the new vector has been created, the necessary information is downloaded to the memory controller.

ams_mapshadow returns 0 on success, and -1 otherwise. Figure 5 is a simple code fragment illustrating the use of ams_mapshadow(AMS_TYPE_PAGECOLOR, ...).

ERRORS:

EINVAL	vaddr is not paged aligned.

2.1.6 ams_mapshadow(AMS_TYPE_VINDIRECT2, datablock)

```
typedef struct {
```

```
/*
 * sum = sparse matrix A times vector P.
 */
foo(float *A, float *P, int *colidx, int *rows,
    int acount, int pcount, float *sum)
{
   vindirect_args_t  vi_args;
   float             *pnew;
   int               i, k;

#ifdef IMPULSE
   vi_args.newaddr    = &pnew;
   vi_args.vaddr      = P;
   vi_args.count      = pcount;
   vi_args.objsize    = sizeof(float);
   vi_args.iv_vaddr   = colidx;
   vi_args.iv_count   = acount;
   vi_args.iv_objsize = sizeof(int);
   vi_args.isfortran  = 0;
   vi_args.maxcount   = acount;
   vi_args.prefcount  = 1;
   vi_args.prefinfo   = 1;
   vi_args.salign     = 0; /* don't care */
   vi_args.valign     = 0; /* don't care */

   if (ams_mapshadow(AMS_TYPE_VINDIRECT, &vi_args) == -1) {
      perror("ams_mapshadow failed.");
      exit(1);
   }
#endif

   for (i = 0; i < SIZE; i++) {
      for (k = rows[i]; k < rows[i+1]; k++)
#ifdef IMPULSE
         sum[k] = A[k] * pnew[k];
#else /* Non-Impulse version */
         sum[k] = A[k] * p[colidx[k]];
#endif
   }
}
```

Figure 4: Code fragment illustrating `ams_mapshadow(AMS_TYPE_VINDIRECT, ...)` usage.

```
/*
 * Map A to the second half of L2 cache and B to the first half
 * to avoid A[i] and B[i] mapping to the same L2 cache line.
 */
foo(double *A, double *B, int size)
{
   int      i;
   double   sum = 0;

#ifdef IMPULSE
   color_array(A, size * sizeof(double), 2, 1);
   color_array(B, size * sizeof(double), 2, 0);
#endif

   for (i = 0; i < SIZE; i++)
      sum += A[i] + B[i];
}

#ifdef IMPULSE
color_array(void *x, int size, int colorfactor, int colorid)
{
   pagecolor_args_t args;

   args.vaddr       = (unsigned) x;
   args.size        = size;
   args.waysize     = 128 * 1024;   /* two-way 256Kbytes L2 cache */
   args.colorfactor = colorfactor;
   args.colorid     = colorid;
   args.prefcount   = 1;
   args.prefinfo    = 1;

   if (ams_mapshadow(AMS_TYPE_PAGECOLOR, &args) == -1) {
      printf("ams_mapshadow_pagecolor failed\n");
      exit(1);
   }
}
#endif
```

Figure 5: Code fragment illustrating `ams_mapshadow(AMS_TYPE_PAGECOLOR, ...)` usage.

```
        unsigned        offset;
        unsigned        size;
} ATTRIB;
#define MAX_ATTR        8

typedef struct {
        unsigned *newaddr;
        unsigned  vaddr;
        int       count;
        int       size;
        unsigned  iv_vaddr;
        int       iv_objsize;
        int       prefcount;
        int       prefinfo;
        int       attribs_num;
        ATTRIB    attribs[MAX_ATTR];
} vindirect2_args_t;
```

The type AMS_TYPE_VINDIRECT2 implements **scatter/gather through an offset vector.** It was specifically designed for the PostSQL database. In PostSQL, the most important data structures are active pages, each of which has the same format: a small header, followed by an offset vector, followed by database records. The length of the last attribute of a record is variable, so the records have variable sizes. The offset vector stores each record's byte offset from the base of the active page. This call maps the required attributes of records in an active page into a dense shadow region. The argument datablock should point to the address of a vindirect2_args_t structure.

newaddr should be a pointer to a location in the application where the kernel will store its return value. vaddr is the page-aligned virtual address of an active page, whose size is size bytes. count represents the number of records in the active page. iv_vaddr is the virtual address of an offset vector, the size of whose object is iv_objsize bytes. attribs_num is the number of requested attributes, with 8 as its maximum value. The offsets and sizes of these required attributes are stored in array attribs[]. prefcount indicates how many lines to prefetch ahead; and prefinfo indicates the prefetch direction and prefetch stride.

The return value of this function is the virtual address of a new vector, which contains count number of elements constituted by only the requested attributes. This new vector is mapped to an equally sized region of shadow address space, but no real pages of physical memory are allocated.

Once the new vector has been created, the necessary information is downloaded to the memory controller.

ams_mapshadow returns 0 on success, and -1 otherwise. If successful, the virtual address of the new vector is placed into the memory location pointed to by newaddr. Figure 6 is a simple code fragment illustrating the use of ams_mapshadow(AMS_TYPE_VINDIRECT2, ...).

ERRORS:

EINVAL	vaddr is not paged aligned.
EFAULT	newaddr specifies an invalid address.

```
typedef struct {
   int    a;
   float  b;
   int    c;
   float  d;
} RECORD;
typedef struct {
   int    a;
   float  d;
} OBJECT;
/*
 * Calculate the sum of "a"/"d".
 */
float foo(RECORD *A, int *offsets, int asize, int objcount,
          int *suma, float *sumd)
{
   vindirect2_args_t  vi2_args;
   OBJECT             *objects;
   RECORD             *record;
#ifdef AMS_TEST
   vi2_args.newaddr           = &objects;
   vi2_args.vaddr             = A;
   vi2_args.count             = objcount;
   vi2_args.size              = asize;
   vi2_args.attribs_num       = 2;
   vi2_args.attribs[0].offset = 0;
   vi2_args.attribs[0].size   = sizeof(int);
   vi2_args.attribs[1].offset = sizeof(int) * 2 + sizeof(float);
   vi2_args.attribs[1].size   = sizeof(float);
   vi2_args.offset_vaddr      = (unsigned) offsets;
   vi2_args.offset_objsize    = sizeof(int);
   vi2_args.prefcount         = 1;
   vi2_args.prefinfo          = 1;

   if (ams_mapshadow(AMS_TYPE_VINDIRECT2, &vi2_args) == -1) {
      perror("ams_mapshadow failed.");
      exit(1);
   }
   for (int i = 0; i < objcount; i++) {
       *suma += objects[i].a; *sumd += objects[i].d;
   }
#else
   for (int i = 0; i < objcount; i++) {
      record = (RECORD *) ((unsigned) A + (unsigned) offsets[i]);
      *suma += record->a; *sumd += record->d;
   }
#endif
}
```

Figure 6: Code fragment illustrating `ams_mapshadow(AMS_TYPE_VINDIRECT2, ...)` usage.

2.2 ams_allocvirt()

```
unsigned long
ams_allocvirt(int size,
              int alignment);
```

User applications can use this system call and `ams_mapvtov()` to optimize their data layout in virtually indexed caches. `ams_allocvirt` allocates a new virtual address range for subsequent use in `ams_mapvtov()` (see Section 2.3). The new address range is not mapped to any physical address space, so it cannot be used until it has be mapped to a region of shadow address space with `ams_mapvtov()`.

`ams_allocvirt` returns the base of the new virtual address range on success, and -1 otherwise.

ERRORS:

`EINVAL`	Either `size` or `alignment` is not page-aligned.

2.3 ams_mapvtov()

```
int
ams_mapvtov(unsigned srcvaddr,
            unsigned dstvaddr
            int      size);
```

User applications can use `ams_allocvirt()` and this system call to optimize their data layout in virtually indexed caches, which include L1 caches in most modern computer systems. `ams_mapvtov` remaps a region of virtual address space starting at `srcvaddr`, to a region of shadow address space originally mapped through a region of virtual address space at `dstvaddr`. The length (in bytes) of the region to be remapped is given by `size`. All of `srcvaddr`, `dstvaddr`, and `size` must be page-aligned values. The range of virtual address space specified by `dstvaddr` and `size` must map to a shadow address region previously allocated by a call to `ams_mapshadow()` (see Section 2.1).

It is worth noting that the application can really screw itself with this call, since the kernel will allow any region of the process' virtual address space to be remapped to any region of shadow address space previously allocated by the process.

`ams_mapvtov` returns 0 on success, and -1 otherwise. Figure 7 is a simple code fragment illustrating the usage of `ams_allocvirt()` and `ams_mapvtov()`.

ERRORS:

`EINVAL`	Either `srcvaddr` or `dstvaddr` is not page-aligned.
`EINVAL`	`size` is not page-aligned.
`EINVAL`	`dstvaddr` and `size` do not specify a valid shadow range.

2.4 ams_remapshadow()

```
int
```

```
/*
 * This function maps A, B, and C to the first, second, and third
 * quadrant of L1 cache respectively.
 * Assume A, B, and C have the same size as one-fourth of the size of
 * L1 cache, and each of them maps to a shadow region.
 */
foo(void *A, void *B, void *C, int size)
{
   void *Av, *Bv, *Cv;

   if ((Av = ams_allocvirt(3 * size, L1_CACHE_SIZE) == -1) {
      perror("ams_mapshadow failed.");
      exit(1);
   }

   Bv = Av + size;
   Cv = Bv + size;

   if ((ams_mapvtov(Av, A, size) == -1) ||
       (ams_mapvtov(Bv, B, size) == -1) ||
       (ams_mapvtov(Cv, C, size) == -1)) {
      perror("ams_mapvtov failed.");
      exit(1);
   }
}
```

Figure 7: Using `ams_allocvirt()` and `ams_mapvtov()` to optimize data layout in virtually indexed L1 cache.

```
/*
 * Change "offset" or "stride" of a data structure pointed by
 * "vsaddr" created by a previous call to ams_mapshadow().
 */
foo(double *vsaddr, int value, int flags)
{
   basestride_args_t bs_args;

   bs_args.newaddr = (unsigned *) &vsaddr;

   if (flags & AMS_REMAP_OFFSET)
      bs_args.offset = value;
   if (flags & AMS_REMAP_STRIDE)
      bs_args.stride = value;

   if (ams_remapshadow(AMS_TYPE_BASESTRIDE, &bs_args, flags) != 0) {
      perror("ams_remapshadow failed.");
      exit(1);
   }
}
```

Figure 8: Code fragment illustrating `ams_remapshadow()` usage.

```
ams_remapshadow(int       type,
                void     *datablock,
                int       flags);
```

`ams_remapshadow` allows user applications to adjust the remapping parameters associated with previous calls to `ams_mapshadow()`. The argument `type` should be one of the following types: `AMS_TYPE_BASESTRIDE`, `AMS_TYPE_VINDIRECT`, or `AMS_TYPE_TRANSPOSE`. Consequently, the argument `datablock` should point to one of the following structures: `basestride_args_t`, `vindirect_args_t`, or `transpose_args_t`.

The `newaddr` of `datablock` is the virtual address returned by a previous call to `ams_mapshadow()`. It allows the kernel to find the proper shadow descriptor.

The `flags` argument specifies what kind of changes to make. If `flags` is `AMS_REMAP_STRIDE`, the new `stride` value contained in the datablock is downloaded to the memory controller. If `flags` is `AMS_REMAP_OFFSET`, the new `offset` value contained in the datablock is downloaded to the memory controller. If `flags` is `AMS_REMAP_PURGE`, kernel purges the data of the specified shadow region out of CPU caches. If `flags` is `AMS_REMAP_FLUSH`, kernel flushes the data of the specified shadow region out of CPU caches.

`ams_remapshadow` returns 0 on success, and -1 otherwise. Figure 8 is a simple code fragment illustrating the usage of `ams_remapshadow()`.

ERRORS:

`EINVAL`	`newaddr` does not specify a valid shadow range.